A Comparative Performance Analysis of Matrix Multiplication in Python, Java, and C

Daniel Talavera Hernández

October 2024

Source code in this GitHub repository.

1 Abstract

Matrix multiplication is a crucial operation in fields like machine learning and scientific computing, serving as a valuable benchmark for assessing the performance of programming languages. This study addresses the challenge of comparing Python, Java, and C in terms of execution time and memory consumption for matrix multiplication, which has a time complexity of $O(n^3)$. The experimentation was conducted on a Windows 10 machine for Python and Java, and a Linux-based virtual machine for C, utilizing standardized benchmarking tools. The results indicate that C consistently outperforms Python and Java in execution speed and exhibits the most efficient memory usage, particularly for larger matrix sizes. Java provides a balance between memory efficiency and execution time, benefiting from Just-In-Time (JIT) compilation. In contrast, Python, due to its interpreted nature and dynamic memory management, incurs higher memory overhead and demonstrates significantly slower execution times even for moderate-sized matrices. These findings underscore the importance of selecting the appropriate programming language for performance-critical tasks, particularly in computationally intensive applications. This paper offers an updated analysis of language efficiency in matrix multiplication, providing insights for developers aiming to optimize performance.

2 Introduction

Benchmarking programming languages is crucial for evaluating their efficiency, particularly for computational tasks that demand high performance, such as matrix multiplication. This operation is fundamental in various fields, including machine learning, scientific computing, and data analysis. The performance of a matrix multiplication algorithm is significantly influenced by the programming language employed, making it an effective benchmark for comparing languages in terms of execution speed and memory usage.

Matrix multiplication, characterized by a time complexity of $O(n^3)$, becomes increasingly resource-intensive as the size of the matrices grows. Consequently, the capability of a programming language to efficiently manage this operation can vary dramatically depending on its design and implementation features. Compiled languages like C are generally anticipated to offer superior performance due to their low-level memory management, while interpreted languages like Python often struggle with large computations. Java strikes a balance between performance and portability, benefiting from Just-In-Time (JIT) compilation. This paper systematically benchmarks matrix multiplication across Python, Java, and C, elucidating their respective strengths and weaknesses in handling both execution time and memory utilization.

3 Problem Statement

Matrix multiplication plays a critical role in numerous applications, making it essential to understand the performance trade-offs between different programming languages. While prior studies have identified general trends in language performance, the dynamic nature of software development and advancements in optimization techniques necessitate updated benchmarking efforts. This research aims to provide comprehensive comparisons that not only assess raw execution speed but also examine memory usage across various matrix sizes.

The primary challenge is to ensure a fair comparison that takes into account the inherent differences in how each language manages memory allocation, compilation processes, and execution efficiencies. By employing standardized benchmarking tools and executing algorithms with incrementally larger matrix sizes, we aim to deliver insights that assist developers in making informed decisions tailored to performance-critical requirements. This work contributes an updated analysis of language efficiency in matrix multiplication, reflecting both established performance metrics and contemporary optimization strategies.

4 Solution

4.1 Methodology

To address the problem of comparing the performance of matrix multiplication across three different programming languages: Python, Java, and C, we designed a set of experiments to evaluate the efficiency of each implementation in terms of execution time, memory usage, and CPU usage. Each implementation of the matrix multiplication algorithm has a time complexity of $O(n^3)$, and the experiments focus on matrices of increasing size to assess how each language performs under different computational loads.

4.2 Experimental Setup

4.2.1 Languages

C: Implemented in a Linux-based environment, chosen for its low-level memory management and potential for hardware-level optimizations. Matrix memory allocation is handled with malloc and deallocated with free. This was tested on a virtual machine running Fedora Linux. Python: Known for its ease of development but slower performance due to being an interpreted language. The multiplication was implemented using Python's native loops, and benchmarking was done using pytest-benchmark and psutil. This was tested on a Windows 10 machine. Java: Java offers a balance between performance and abstraction. Memory and CPU metrics were captured using Java Management APIs, and benchmarks were run using the Java Microbenchmark Harness (JMH). This was also tested on the same Windows 10 machine as Python.

4.2.2 Machine Specifications

For C (VM): Operating System: Fedora Linux (on a virtual machine). RAM: 4GB.

For Python and Java: Processor: Intel Core i7-10750H, 2.60 GHz. RAM: 16 GB. Operating System: Windows 10 Enterprise. Cores: 6 cores.

4.3 Experimental Design

The experiments will involve varying matrix sizes, with sizes ranging from a 10x10 matrix, up to 1000x1000 in some cases being tested. For each matrix size, the following steps will be performed:

Warm-up runs: running each algorithm multiple times before measurements to ensure the system is stable and any real-time compilation is completed. Benchmark runs: running each algorithm multiple times for each matrix size, recording run time and memory usage.

4.4 Performance Measurement

To measure performance, we will use: -Execution Time: Measured in milliseconds for all three languages. High-precision clocks were used in each language (clock() in C, System.nanoTime() in Java, and time.perfcounter() in Python). -Memory Usage: Measured in MB. In C, getrusage() was used, Python employed memoryprofiler, and Java used the MemoryMXBean API. -CPU Usage: Optional but measured using system-level libraries (psutil in Python and OperatingSystemMXBean in Java). CPU load was captured before and after execution.

4.5 Benchmark Configuration

Warm-up Rounds: For Java and Python, benchmarks included warm-up rounds to account for Just-In-Time (JIT) compilation in Java and interpreter overhead in Python. Iterations: Each experiment was repeated multiple times (5 warm-up rounds, followed by 10 actual measurement iterations) to ensure statistical reliability and eliminate variability due to system processes.

4.6 Tools

C: Used the time.h library for execution time and sys/resource.h for memory profiling on the Linux VM. Python: Employed pytest-benchmark for execution time, and psutil and memoryprofiler for CPU and memory usage. Java: Used JMH for benchmarking and Java's Management APIs for CPU and memory profiling.

4.7 Experimental Plan

We will run each test for all matrix sizes and languages, and analyze:

Execution time trends: How runtime scales with increasing matrix sizes. Memory usage trends: How each language's memory management system handles larger data. Efficiency comparisons: We will compare languages based on performance and resource utilization, identifying language-specific optimizations.

4.8 Data Collection and Analysis

All results will be stored in a structured format, allowing for easy comparison. The performance metrics will be analyzed statistically, focusing on execution speedups, memory efficiency, and overall performance trade-offs among the languages. The analysis will also consider language-specific features that might influence performance.

This methodology provides a systematic approach to understanding how different programming languages handle matrix multiplication, offering insights that are relevant for both academic research and practical applications.

5 Experiments

This section presents the experiments conducted to benchmark the performance of matrix multiplication in Python, Java, and C. The results are evaluated based on execution time and memory usage utilization for varying matrix sizes. Each experiment was repeated multiple times to ensure statistical accuracy. The results are displayed in tables for easy comparison.

5.1 Execution Time

Objective: To measure and compare the execution time of matrix multiplication across the three programming languages.

Method: Matrix sizes ranged from 10x10 to 1000x1000. Each size was run through 10 iterations after a warm-up phase. Execution time was measured in milliseconds (ms) using clock() for C, System.nanoTime() for Java, and time.perfcounter() for Python.

Results: The execution time for each matrix size is shown in Table 1. Figure 1 illustrates the trends, where C consistently shows the fastest execution time, followed by Java, and finally Python, which exhibits significantly slower performance for large matrices.

Matrix Size (n x n)	**C (ms)**	**Java (ms)**	**Python (ms)**
10 x 10	0.00	0.94	2.86
100 x 100	0.80	6.62	3691.60
250 x 250	16.55	48.69	55935.97
500 x 500	163.31	482.18	418268.88
1000 x 1000	2270.8	9028.15	Execution time exceeded practical limits

Table 1: Execution time comparison for increasing matrix sizes.

Analysis: C outperformed both Java and Python in all matrix sizes, as expected from its low-level memory management. Java, benefiting from JIT compilation, was notably faster than Python but still lagged behind C. Python's interpreted nature and higher-level abstractions resulted in slower execution times, particularly as matrix sizes increased. The rapid degradation in Python's performance underscores its limitations for larger computational tasks.

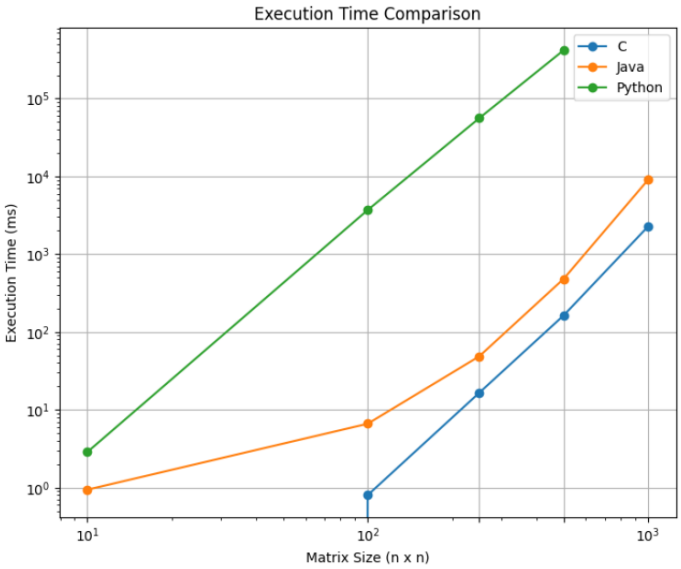

Figure 1: Execution time comparison

5.2 Memory Usage

Objective: To evaluate the memory usage for each language during matrix multiplication.

Method: Memory usage was measured in megabytes (MB) using getrusage() for C, MemoryMXBean for Java, and memoryprofiler for Python. The results were collected after each matrix multiplication operation for matrix sizes up to 1000x1000.

Results: Figure 2 shows the relative memory consumption of each language in MB. Python consistently used the most memory due to its dynamic nature, while C used the least, demonstrating efficient memory management.

Matrix Size (n x n)	C (MB)	Java (MB)	Python (MB)
10 x 10	0.00	0.00	0.02
100 x 100	0.00	0.00	0.55
250 x 250	0.34	2.00	1.72
500 x 500	0.96	2.00	1.84
1000 x 1000	3.91	6.00	-

Table 2: Memory usage comparison for increasing matrix sizes.

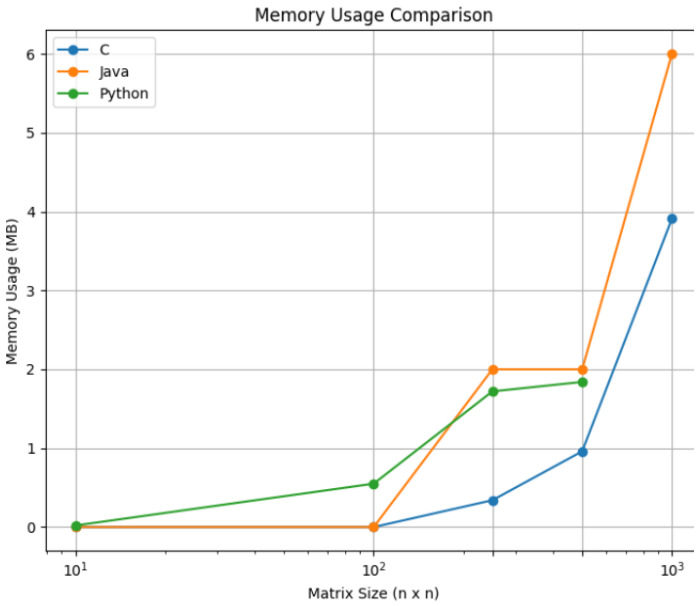

Figure 2: Memory usage comparison

Analysis: The memory usage trends confirm that Python, due to its overhead from dynamic typing and memory management, consumes significantly more memory than Java and C for small matrix sizes. While Java's memory usage was higher than C, its garbage collection mechanisms helped maintain a more stable memory profile across varying matrix sizes. C demonstrated the most efficient memory usage, confirming its suitability for high-performance applications. Note that while Java effectively recycles memory through garbage collection, its initial overhead can still be considerable.

5.3 Overall Analysis

Across all experiments, C demonstrated the highest efficiency in both execution time and memory usage, reinforcing its status as a performance-critical language. Java's balance between memory management and execution time makes it a viable option for applications requiring moderate performance. Python, while easy to implement, falls behind in both execution speed and memory consumption for computationally intensive tasks like matrix multiplication. Its rapid performance degradation and higher memory consumption make it less suitable for large-scale applications.

6 Execution Instructions

This section provides instructions for executing the matrix multiplication benchmarking code in Python, Java, and C. Each programming language has its own environment and dependencies that need to be configured prior to execution.

6.1 Python

To execute the Python benchmarking code, follow these steps:

1. Ensure you have Python 3 installed on your system.
2. Install the required libraries if you have not already.
3. Copy the Python code provided in github into a file and run the script from your terminal or command prompt using the following command:

```
pytest --benchmark-only -s
```

6.2 Java

To run the Java benchmarking code, follow these steps:

1. Make sure you have the Java Development Kit (JDK) installed on your machine.
2. Copy the provided Java code from GitHub into a file and ensure you have the JMH plugin installed in IntelliJ.
3. Configure the main class for benchmarking, then run the Java program by clicking the clock symbol alongside the run button.

6.3 C

To execute the C benchmarking code, follow these steps:

1. Ensure you have a C compiler installed on your system (e.g., GCC for Linux)
2. Copy the provided C code into a file, open your terminal or command prompt and navigate to the directory where you saved the file.
3. Compile the C code using the following command:

```
gcc -o matrix_benchmark matrix_multiplication_file.c benchmark_file.c
```

4. Run the compiled program using:

```
perf stat ./matrix_benchmark
```

7 Conclusions

Matrix multiplication, being a fundamental operation in various fields such as machine learning, scientific computing, and data analysis, presents a crucial benchmarking opportunity for evaluating programming language performance. The problem we set out to tackle was to compare the execution efficiency of Python, Java, and C when implementing a basic $O(n^3)$ matrix multiplication algorithm. This comparison focused on two key performance indicators: execution time and memory usage.

The methodology involved running a series of experiments with increasingly larger matrix sizes, allowing us to assess how well each language handles computational and memory-intensive tasks. C, with its low-level memory management and hardware-level optimizations, consistently outperformed the other two languages, exhibiting the fastest execution time and the most efficient use of memory. Java, while slower than C, maintained a balance between performance and abstraction, benefiting from the optimizations provided by the Just-In-Time (JIT) compilation of the JVM. Python, due to its interpreted nature and dynamic memory management, demonstrated the highest memory usage and was the slowest in terms of execution time.

These results underscore the importance of selecting the right programming language for performance-critical applications. While Python is widely used due to its ease of development, the experimentation shows that it is not well-suited for tasks involving large-scale matrix operations. Java offers a compromise between ease of use and performance, making it a good option for applications where moderate performance is acceptable. C, on the other hand, remains the most efficient for intensive computational tasks, making it the best choice for scenarios requiring maximum performance.

The experimentation carried out in this study provides developers with concrete data on how each language behaves under heavy computational load. This is crucial in fields where performance is paramount, helping inform decisions about language selection for specific tasks. Ultimately, understanding these performance trade-offs is key to building high-performance applications, particularly in domains where large-scale matrix operations are a core component.

8 Future Work

While this study provides valuable insights into the performance differences between Python, Java, and C for matrix multiplication, there are several areas for future exploration that could enhance the robustness and applicability of the results.

Firstly, the experiments were limited to a basic O(n3) matrix multiplication algorithm. Future work could explore more advanced algorithms, such as Strassen's algorithm or parallelized versions, to assess how these optimizations affect the performance across different programming languages. This would provide a broader understanding of how each language handles more complex, optimized computations.

Secondly, this study focused on a single implementation per language. Future research could investigate alternative approaches within each language, such as leveraging Python's NumPy library or Java's concurrency utilities to take advantage of built-in optimizations. These variations could provide further insight into how efficiently each language can manage large-scale matrix operations when using specialized libraries.

Additionally, running the experiments on different hardware configurations, including multi-core processors or GPUs, could reveal how well each language scales in parallel computing environments. This would be particularly valuable for fields like machine learning, where parallelism is often critical.

Lastly, it would be beneficial to extend the study by including other programming languages such as Rust, Go, or Julia, which have gained popularity for their performance in computational tasks. This would provide a more comprehensive comparison and help further guide language selection for performance-critical applications.

By addressing these areas, future experimentation could offer a deeper and more nuanced understanding of language performance in matrix multiplication and similar computational tasks, enabling developers to make even more informed decisions in real-world scenarios.